А. И. ЕЛИСЕЕВ, Ю. В. МИНИН

РАЗРАБОТКА ПРОГРАММНЫХ ИНТЕРФЕЙСОВ ВЕБ-ПРИЛОЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ ФРЕЙМВОРКА FastAPI

Тамбов
Издательский центр ФГБОУ ВО «ТГТУ»
2024

Министерство науки и высшего образования Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования «Тамбовский государственный технический университет»

А. И. ЕЛИСЕЕВ, Ю. В. МИНИН

РАЗРАБОТКА ПРОГРАММНЫХ ИНТЕРФЕЙСОВ ВЕБ-ПРИЛОЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ ФРЕЙМВОРКА FastAPI

Утверждено Ученым советом университета
в качестве учебного пособия для студентов 2, 3 курсов,
обучающихся по направлению подготовки 09.03.02
«Информационные системы и технологии»

Учебное электронное издание

Тамбов
Издательский центр ФГБОУ ВО «ТГТУ»
2024

УДК 004(075.8)
ББК *з*973.43я73
Е51

Рецензенты:

Кандидат технических наук, доцент, доцент Института новых технологий и искусственного интеллекта ФГБОУ ВО «ТГУ им. Г. Р. Державина»
И. А. Зауголков

Кандидат технических наук, доцент кафедры «Мехатроника и технологические измерения» ФГБОУ ВО «ТГТУ»
А. С. Егоров

Елисеев, А. И.

Е51 Разработка программных интерфейсов веб-приложений с использованием фреймворка FastAPI [Электронный ресурс] : учебное пособие / А. И. Елисеев, Ю. В. Минин. – Тамбов : Издательский центр ФГБОУ ВО «ТГТУ», 2024. – 1 электрон. опт. диск (CD-ROM). – Системные требования : ПК не ниже класса Pentium II ; CD-ROM-дисковод ; 1,5 Mb ; RAM ; Windows 95/98/XP ; мышь. – Загл. с экрана.

ISBN 978-5-8265-2821-1

Представляет собой руководство по использованию фреймворка FastAPI, охватывающее основные концепции и практические примеры для создания веб-приложений на Python. Рассмотрены ключевые аспекты работы с FastAPI, включая маршрутизацию, обработку запросов и ответов, валидацию данных, аутентификацию, а также интеграцию с базами данных и асинхронное программирование.

Предназначено для студентов 2, 3 курсов, обучающихся по направлению подготовки 09.03.02 «Информационные системы и технологии».

УДК 004(075.8)
ББК *з*973.43я73

ISBN 978-5-8265-2821-1

ВВЕДЕНИЕ

FastAPI – современный, высокопроизводительный веб-фреймворк для создания программных интерфейсов приложений (Application Programming Interface, API) на языке программирования Python. Он был разработан с акцентом на скорость, простоту использования и поддержку асинхронного подхода. FastAPI использует стандартные Python-аннотации типов для автоматической генерации документации и валидации данных, что значительно упрощает процесс разработки.

Преимущества FastAPI

1. Высокая производительность: FastAPI построен на основе ASGI (Asynchronous Server Gateway Interface) и использует асинхронные возможности Python, что позволяет ему обрабатывать большое количество запросов с минимальными задержками. По производительности он сопоставим с такими фреймворками, как Node.js и Go.

2. Простота и удобство: Благодаря использованию аннотаций типов FastAPI автоматически генерирует документацию для API, что облегчает его тестирование и интеграцию.

3. Автоматическая валидация данных: FastAPI автоматически проверяет входные данные на соответствие заданным типам и схемам, что повышает надежность и безопасность приложения.

4. Совместимость с современными стандартами: FastAPI полностью поддерживает OpenAPI и JSON Schema, что делает его идеальным выбором для создания RESTful API и микросервисов.

5. Широкая экосистема: FastAPI легко интегрируется с другими популярными библиотеками и инструментами, такими как SQLAlchemy, Pydantic, и Jinja2, что позволяет создавать комплексные и многофункциональные приложения.

В пособии рассмотрены основные концепции и возможности FastAPI, рассмотрен процесс создания программных интерфейсов асинхронных веб-приложений.

ВЕБ-ФРЕЙМВОРКИ НА Python

Web Server Gateway Interface (WSGI) – это спецификация стандарта синхронизации (https://peps.python.org/pep-3333) Python для подключения кода приложения к веб-серверам. Все традиционные веб-фреймворки Python построены на WSGI. Но синхронное взаимодействие часто является медленным. В результате развития WSGI была разработана спецификация Asynchronous Server Gateway Interface (ASGI) для Python.

Популярные фреймворки на Python, поддерживающие WSGI:

– Django;

– Flask.

Разработчики Flask называют его микрофреймворком. Он предоставляет базовые возможности, все остальное – доступно в загружаемых сторонних пакетах. Он меньше и проще, чем Django, и его можно быстрее освоить. Flask относится к типу синхронных фреймворков и создан на базе стандарта WSGI, а не ASGI.

Django – это более крупный и сложный проект, чем Flask или FastAPI. На Django можно реализовать проект любой сложности. Django имеет встроенный ORM (Object Relational Mapping). Хотя Django был традиционным WSGI-приложением, в версии 3.0 была добавлена поддержка стандарта ASGI.

FastAPI – это современный веб-фреймворк для создания API на Python, который отличается высокой производительностью и простотой использования. Он использует ASGI для поддержки асинхронных операций, что позволяет обрабатывать множество запросов одновременно и повышает общую эффективность приложения. Для запуска приложений на FastAPI часто используется Uvicorn – быстрый ASGI-сервер, написанный на Python, который обеспечивает высокую производительность и низкую задержку при обработке запросов.

В FastAPI используются:

– подсказки типов Python;

– пакет Starlette, включая поддержку асинхронности;

– пакет Pydantic для определения и проверки данных;

– возможности интеграции, позволяющие использовать и расширять возможности фреймворка.

Необходимые пакеты для работы с FastAPI:

– фреймворк FastAPI – `poetry add fastapi`;

– веб-сервер Uvicorn – `poetry add uvicorn`;

Дополнительно в примерах пособия используются:

– веб-клиент HTTPie – `poetry add httpie`;

– пакет синхронного веб-клиента Requests – `poetry add requests`;

– пакет синхронного/асинхронного веб-клиента HTTPX – `poetry add httpx`.

ВВЕДЕНИЕ В FastAPI

Первое приложение

Первый пример:

```python
# main.py
from fastapi import FastAPI

app = FastAPI()

@app.get("/hi")
def greet():
    return "Hello World!"
```

app – объект FastAPI верхнего уровня, представляющий все веб-приложение.

@app.get("/hi") – декоратор пути, обрабатывает запрос к URL-адресу /hi, применяется только к HTTP-глаголу GET.

def greet() представляет собой функцию пути – основную точку контакта с HTTP-запросами и ответами.

Запуск приложения с помощью командной строки:

```
uvicorn main:app --reload
```

main – ссылка на файл main.py, а app – имя объекта FastAPI в файле.

Запуск Uvicorn внутри приложения:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/hi")
def greet():
    return "Hello World!"

if __name__ == "__main__":
    import uvicorn
    uvicorn.run("main:app", reload=True)
```

Параметр reload нужен для перезапуска веб-сервера, если содержимое файла main.py изменится.

Для тестирования конечной точки /hi нужно перейти по ссылке http://127.0.0.1:8000/hi.

Проверка с помощью Requests:

```
>>> import requests
>>> r = requests.get("http://127.0.0.1:8000/hi")
>>> r.json()
'Hello World!'
```

Проверка с помощью HTTPX:

```
>>> import httpx
>>> r = httpx.get("http://127.0.0.1:8000/hi")
>>> r.json()
'Hello World!'
```

Проверка с помощью HTTPie:

```
http 127.0.0.1:8000/hi

HTTP/1.1 200 OK
content-length: 15
```

```
content-type: application/json
date: Thu, 30 Jun 2024 07:38:27 GMT
server: uvicorn
"Hello World!"
```

Проверка с помощью HTTPie с выводом только тела ответа:

```
http -b 127.0.0.1:8000/hi "Hello World!"
```

Проверка с помощью HTTPie с получением всех данных:

```
http -v 127.0.0.1:8000/hi

GET /hi HTTP/1.1
Accept: /
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: 127.0.0.1:8000
User-Agent: HTTPie/3.2.1

HTTP/1.1 200 OK
content-length: 15
content-type: application/json
date: Thu, 30 Jun 2024 08:05:06 GMT
server: uvicorn

"Hello World!"
```

Типичный запрос содержит:

– глагол-оператор (GET) и путь (/hi);

– все параметры запроса (текст после символа `?` в данном случае отсутствует);

– другие HTTP-заголовки;

– содержимое тела запроса (отсутствует).

Способы передачи параметров:

– в пути URL;

– в качестве параметра запроса после символа `?` в URL;

– в теле HTTP-сообщения;

– в HTTP-заголовке.

Использование пути:

```
from fastapi import FastAPI

app = FastAPI()

@app.get("/hi/{who}")
def greet(who):
    return f"Hello {who}!"
```

Добавление {who} в URL-адрес (после выражения @app.get) приводит к извлечению переменной с именем who в указанном местоположении в URL.

Затем FastAPI присваивает ее аргументу who в функции greet().

Не следует использовать f-строку для строки URL ("/hi/{who}"). Фигурные скобки применяются самим FastAPI для использования частей URL в качестве параметров пути.

Проверка в браузере: 127.0.0.1:8000/hi/user.

Проверка с помощью HTTPie:

```
http 127.0.0.1:8000/hi/user

HTTP/1.1 200 OK
content-length: 13
content-type: application/json
date: Thu, 30 Jun 2024 08:09:02 GMT
server: uvicorn

"Hello user!"
```

Параметры запроса – это выражения вида name=value после символа ? в URL-адресе, разделенные символами &.

```
@app.get("/hi")
def greet(who):
    return f"Hello {who}!"
```

Проверка с помощью браузера: 127.0.0.1:8000/hi?who=user.

Проверка с помощью HTTPie:

```
http -b 127.0.0.1:8000/hi?who=user

"Hello user!"
```

Или так:

```
http -b 127.0.0.1:8000/hi who==user

"Hello user!"
```

Конечной точке GET можно предоставить путь или параметры запроса, но не значения из тела запроса. В HTTP GET-запрос должен быть идемпотентным.

Тело запроса используется для отправки данных на сервер при операциях создания (POST) или обновления (PUT или PATCH).

Пример:

```
@app.post("/hi")
def greet(who: str = Body(embed=True)):
    return f"Hello {who}!"
```

Выражение Body(embed=True) определяет, что на этот раз мы получаем значение who из тела запроса в формате JSON. Часть выражения embed в скобках означает, что запрос должен выглядеть как {"who": "user"}, а не просто "user".

Проверка с помощью HTTPie:

```
http -v 127.0.0.1:8000/hi who=user

POST /hi HTTP/1.1
Accept: application/json, /;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 14
Content-Type: application/json
Host: 127.0.0.1:8000
User-Agent: HTTPie/3.2.1

{
  "who": "user"
}


HTTP/1.1 200 OK
content-length: 13
content-type: application/json
date: Thu, 30 Jun 2024 08:37:00 GMT
server: uvicorn

"Hello user!"
```

Параметры запроса

Параметры запроса (Request Parameters) – специальные параметры в функции, декорированной операциями пути, которые позволяют получать данные из запроса.

Параметры запроса разрешаются путем внедрения зависимости.

FastAPI любой запрос разложит по следующим объектам:

– Header – HTTP-заголовки;

– Path – URL-адрес;

– Query – параметры запроса (после символа ? в конце URL);

– Body – тело HTTP-сообщения.

В приведенном ниже запросе есть строки запроса skip и limit:

```
http://127.0.0.1:8000/items/?skip=0&limit=103
```

Приведенный ниже код FastAPI определяет функцию read_items(), которая отвечает на GET-запросы к конечной точке /items с двумя параметрами skip и limit:

```
from fastapi import Query
...
@app.get("/items")
async def read_items(
        skip: int = Query(default=0, gt=0),
        limit: int = Query(default=10, lt=100)):
        return {"skip":skip, "limit":limit}
```

Параметры запроса, объявленные в функции, типизируются с помощью объекта Query. Query инстанцируется с тремя параметрами: default, gt и lt. Все параметры запроса имеют набор параметров, которые могут быть использованы для обогащения документации или применения валидации.

Тестирование:

```
http -v http://127.0.0.1:8000/items skip==10 limit==10

GET /items?skip=10&limit=10 HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
```

```
Connection: keep-alive
Host: 127.0.0.1:8000
User-Agent: HTTPie/3.2.2


HTTP/1.1 200 OK
content-length: 22
content-type: application/json
date: Mon, 12 Aug 2024 18:27:14 GMT
server: uvicorn

{
    "limit": 10,
    "skip": 10
}
```

Параметры пути – это значение в пути URL, которое идентифицирует определенный ресурс или набор ресурсов на сервере, например, ID пользователя.

Пример передачи параметров в пути:

```
http://127.0.0.1/users/1
```

Реализация:

```
from fastapi import Path
...
@app.get("/users/{id}")
async def read_item(
    id: int = Path(gt=0,
                   title="User id",
                   description="User identifier"),):
                   return {"id": id}
```

Тестирование:

```
http -v 127.0.0.1:8000/users/1

GET /users/1 HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: 127.0.0.1:8000
User-Agent: HTTPie/3.2.2


HTTP/1.1 200 OK
content-length: 8
```

```
content-type: application/json
date: Mon, 12 Aug 2024 18:23:20 GMT
server: uvicorn

{
    "id": 1
}
```

Параметры заголовка – это значения, используемые для предоставления дополнительной информации между клиентом и сервером во время транзакции. HTTP-заголовок состоит из пар ключ:значение, разделенных двоеточиями (:). Параметры заголовка не отображаются в запросе URL как параметры запроса или пути. Обычно они обрабатываются или регистрируются только сервером или клиентским приложением.

Наиболее распространенными заголовками запросов API являются: Accept, Authorization, Content Type, Cache Control и User Agent.

Приведенный ниже код используется для установки пользовательского HTTP-заголовка, который используется клиентом для передачи серверу API-ключа:

```
from fastapi import Header
...
REGEX_API_KEY = r"^[A-Za-z0-9]{32}$"

@app.get("/secure-data")
async def read_secure_data(api_key: str = Header(...)):
    if not re.match(REGEX_API_KEY, api_key):
        raise HTTPException(status_code=400, detail="Invalid API Key
format")
    return {"message": "Access granted", "api_key": api_key}
```

Тестирование:

```
http http://127.0.0.1:8000/secure-data
api-key:1234567890abcdef1234567890abcdef

GET /secure-data HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: 127.0.0.1:8000
```

```
User-Agent: HTTPie/3.2.2
api-key: 1234567890abcdef1234567890abcdef



HTTP/1.1 200 OK
content-length: 73
content-type: application/json
date: Sun, 08 Sep 2024 16:54:44 GMT
server: uvicorn

{
    "api_key": "1234567890abcdef1234567890abcdef",
    "message": "Access granted"
}
```

Cookie в целом используются для хранения данных на стороне клиента. Часто используются для управления сессией, персонализации (хранение настроек пользователя) или отслеживания поведения пользователя.

В примере название страны, хранящееся в cookie, передается на сервер.

```
from fastapi import Cookie
...
@app.get("/blog")
async def read_items(country: str = Cookie(default=None)):
    return {"Country": country}
```

Добавить cookie для тестирования можно напрямую в консоли разработчика:

```
document.cookie='country=Russia'
```

Параметры тела запроса, иногда называемые полезной нагрузкой, используются, когда клиенту необходимо отправить данные на сервер для создания или обновления ресурсов в POST, PUT или PATCH-запросах.

Тело запроса может быть в различных форматах, включая JSON, XML и обычный текст.

В примере ниже показана конечная точка, которая получает поле тела запроса с названием content в формате обычного текста:

```
from fastapi import Body
...
@app.post("/blog")
async def read_items(content: str = Body(...)):
    return {"Content": content}
```

Тестирование:

```
http -v http://127.0.0.1:8000/blog Content-Type:text/plain
<<< Message

POST /blog HTTP/1.1
Accept: application/json, */*;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 8
Content-Type: text/plain
Host: 127.0.0.1:8000
User-Agent: HTTPie/3.2.2

Message

HTTP/1.1 200 OK
content-length: 23
content-type: application/json
date: Mon, 12 Aug 2024 18:34:10 GMT
server: uvicorn

{
  "Content": "Message\n"
}
```

Параметры формы – это данные, которые передаются через HTML-форму и отправляются на сервер с помощью метода POST или GET.

Использование метода GET небезопасно и не рекомендуется, особенно для конфиденциальной информации, такой как пароли.

Значения отправляются в виде пар ключ-значение, где ключ соответствует имени поля формы, а значение – это данные, введенные пользователем.

Заголовок Content-Type в HTTP-запросе имеет значение application/x-www-form-urlencoded, если используются параметры формы, или multipart/form-data, если форма используется для загрузки файлов.

В примере ниже мы используем форму для отправки имени пользователя и пароля на сервер:

```
from fastapi import Form
...
@app.post("/login")
async def read_items(user: str = Form(...), password: str =
Form(...)):
    return {"user": user, "password": password}
```

Тестирование:

```
http -v --form http://127.0.0.1:8000/login Content-
Type:application/x-www-form-urlencoded user=admin password=secret

POST /login HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 26
Content-Type: application/x-www-form-urlencoded
Host: 127.0.0.1:8000
User-Agent: HTTPie/3.2.2

user=admin&password=secret


HTTP/1.1 200 OK
content-length: 36
content-type: application/json
date: Mon, 12 Aug 2024 18:37:57 GMT
server: uvicorn

{
    "password": "secret",
    "user": "admin"
}
```

Параметры файла позволяют отправить файл как часть запроса на сервер. Тип содержимого запроса – multipart/form-data.

Загрузка файлов – одна из тех задач, которые сложно реализовать в API. К счастью, в FastAPI эта задача решается несложно:

```
from fastapi import File, UploadFile
import uuid
...
@app.post("/upload")
async def read_items(photo: UploadFile = File(...)):
    filename = f"{uuid.uuid4()}-{photo.filename}"

    with open(filename, "wb") as f:
        while contents := photo.file.read(1024 * 1024):
            f.write(contents)

    return {"Photo": filename}
```

Тестирование:

```
http -v --form http://127.0.0.1:8000/upload Accept:application/json
Content-Type:multipart/form-data photo@file.png

POST /upload/ HTTP/1.1
Accept: application/json
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 113068
Content-Type: multipart/form-data; bounda-
ry=80da9cde836247c7b24c43163dcafbc8
Host: 127.0.0.1:8000
User-Agent: HTTPie/3.2.2

+-----------------------------------------+
| NOTE: binary data not shown in terminal |
+-----------------------------------------+

HTTP/1.1 200 OK
content-length: 57
content-type: application/json
date: Mon, 12 Aug 2024 18:41:41 GMT
server: uvicorn

{
    "Photo": "bdb81b15-576e-4993-9308-6c22ff9ac248-file.png"
}
```

В одной функции пути можно использовать более одного из этих методов. Можно получать данные из URL, параметров запроса, тела HTTP, HTTP-заголовков, cookie-файлов и т.д. Можно написать собственные

функции зависимости, которые будут обрабатывать и объединять их особым образом, например, для пагинации или аутентификации.

Какой вариант выбрать? При передаче аргументов в URL стандартной практикой стало следование рекомендациям стиля REST. Строки запросов обычно применяются для предоставления дополнительных аргументов, таких как пагинация. Тело запроса обычно используется для больших объемов, вводимых данных, например, целых или частичных моделей.

HTTP-ответы

По умолчанию FastAPI преобразует все, что возвращается из функции конечной точки, в формат JSON. Поэтому HTTP-ответ содержит строку заголовка Content-type: application/json.

Типы ответов (классы из модуля fastapi.responses):

– JSONRespons (по умолчанию);
– HTMLResponse;
– PlainTextResponse;
– RedirectResponse;
– FileResponse;
– StreamingResponse.

Для других форматов вывода можно использовать общий класс Response, требующий следующие параметры:

– content – строка или байт;
– media_type – строка MIME-типа;
– status_code – целочисленный код состояния HTTP;
– headers – словарь строк.

Код состояния

По умолчанию FastAPI возвращает код состояния *200*. Исключения вызывают коды группы *4xx*.

Указание кода состояния:

```
@app.get("/happy")
def happy(status_code=200):
  return ":)"
```

В FastAPI модуль fastapi.status содержит набор констант, которые соответствуют стандартным кодам статуса HTTP. Использование констант делает код более читаемым и поддерживаемым, так как они предоставляют удобные имена для стандартных кодов статуса.

```
@app.get("/happy")
def happy(status_code=status.HTTP_200_OK):
  return ":)"
```

Заголовки

Можно вводить заголовки HTTP-ответов (не нужно возвращать сообщения response):

```
from fastapi import Response

@app.get("/header/{name}/{value}")
def header(name, value, response:Response):
    response.headers[name] = value
    return "normal body"
```

Проверка HTTP-заголовков ответа:

```
$ http 127.0.0.1:8000/header/marco/polo

HTTP/1.1 200 OK
content-length: 13
content-type: application/json
date: Wed, 31 May 2023 17:47:38 GMT
marco: polo
server: uvicorn

"normal body"
```

Автоматизированная документация

FastAPI генерирует спецификацию OpenAPI из кода и включает эту страницу для отображения и тестирования всех конечных точек.

Документация в виде Swagger UI доступна по адресу http://127.0.0.1:8000/docs.

Есть второй вариант – ReDoc, который доступен по адресу http://127.0.0.1:8000/redoc.

АСИНХРОННОСТЬ

Большая часть веб-кода FastAPI основана на пакете Starlette. Самая важная особенность заключается в поддержке современного асинхронного веб-стандарта Python – ASGI. Большинство веб-фреймворков Python, например Flask и Django, основывались на традиционном синхронном стандарте WSGI. ASGI позволяет избежать характерных для приложений на базе WSGI блокировок.

В результате Starlette и использующие его фреймворки стали самыми быстрыми веб-пакетами Python и составили конкуренцию даже приложениям на Go и Node.js.

В Python async и await используются для работы с асинхронным кодом, позволяя выполнять операции параллельно без блокировки основного потока выполнения.

asyncio – это стандартная библиотека Python для написания асинхронного кода с использованием корутин, задач и событийного цикла.

Операторы async и await

Использование ключевых слов async и await само по себе не ускоряет выполнение кода. На самом деле такой код может оказаться немного медленнее из-за накладных расходов на асинхронную настройку. Основное назначение конструкций async заключается в том, чтобы избежать длительного ожидания ввода-вывода.

Пример асинхронной точки доступа:

```
from fastapi import FastAPI
import asyncio

app = FastAPI()

@app.get("/hi")
async def greet():
    await asyncio.sleep(1)
    return "Hello World!"
```

Использование asyncio.sleep(1) имитирует реальную функцию, занимающую одну секунду, например, вызов базы данных или загрузку веб-страницы.

ВАЛИДАЦИЯ ДАННЫХ

В Python 3.10 добавили подсказки типов к языку в стандартный модуль. Интерпретатор Python игнорирует синтаксис подсказки типа и выполняет программу так, как будто ее нет. Тогда в чем смысл?

Инструменты-помощники, такие как mypy, обращают внимание на подсказки типов и предупреждают о любых несоответствиях. Подсказки доступны разработчикам, которые могут написать инструменты, выполняющие не только проверку ошибок типов. Один из таких инструментов – пакет Pydantic.

Подсказки типов

Существует один синтаксис для переменных и другой – для возвращаемых значений функций. Подсказки типа переменной могут включать только тип:

```
name: type
```

Или также инициализировать переменную значением:

```
name: type = value
```

Тип может быть одним из стандартных простых типов, таких как int или str, или коллекцией, такой как tuple, list или dict:

```
thing: str = "string"
```

При использовании Python до версии 3.9 необходимо импортировать версии стандартных имен типов из модуля типизации:

```
from typing import Str

thing: Str = "string"
```

Несколько примеров с инициализацией:

```
physics_magic_number: float = 1.0/137.03599913
hp_lovecraft_noun: str = "ichor"
exploding_sheep: tuple = "sis", "boom", bah!"
responses: dict = {"Marco": "Polo", "answer": 42}
```

Можно также включать подтипы коллекций:

```
name: dict[keytype, valtype] = {key1: val1, key2: val2}
```

Модуль типизации содержит полезные дополнения для подтипов.

Наиболее распространенные:

– *Any* – любой тип;

– *Union* – любой из указанных типов, например `Union[str, int]`.

Начиная с версии 3.10, можно написать type1 | type2, а не Union[type1, type2].

Примеры определений для словарей в Python 3.9:

```
from typing import Any

responses: dict[str, Any] = {"Marco": "Polo", "answer": 42}
```

Или, если быть более точными:

```
from typing import Union
responses: dict[str, Union[str, int]] = {"Marco": "Polo", "answer":
42}
```

В Python 3.10 и более поздних версиях:

```
responses: dict[str, str | int] = {"Marco": "Polo", "answer": 42}
```

Обратите внимание, что переменная с подсказкой типа является верной, а простая переменная – нет:

```
$ python
...
>>> thing0
Traceback (most recent call last):
File "<stdin>", line 1, in <module> NameError: name thing0 is not
defined
>>> thing0: str
```

Некорректное использование типов не отлавливается обычным интерпретатором:

```
$ python
...
>>> thing1: str = "yeti"
>>> thing1 = 47
```

Но такие ошибки будут обнаружены с помощью статического анализатора mypy.

Установка mypy:

```
poetry add -G dev mypy
```

Проверка:

```
mypy stuff.py
stuff.py:2: error: Incompatible types in assignment (expression has
type "int", variable has type "str") Found 1 error in 1 file (checked
1 source file)
```

В подсказке типа возврата функции вместо двоеточия применяется стрелка:

```
function(args) -> type:
```

Пример:

```
def get_thing() -> str:
    return "yeti"
```

Можно задействовать любой тип, включая определенные классы или их комбинации.

Структуры данных

Структуры данных в Python (помимо базовых *int*, *string* и подобных им):

– tuple – кортеж, неизменяемая последовательность объектов;

– list – список, изменяемая последовательность объектов;

– set – множество, изменяемые уникальные объекты;

– dict – словарь, пары изменяемых объектов «ключ – значение» (ключ должен быть неизменяемого типа).

Кортежи и списки позволяют обращаться к переменной по индексу:

```
tuple_movie = ("Inception", "Christopher Nolan", "2010-07-16", 148,
"Science Fiction", 8.8)
print("Title is", tuple_movie[0])

list_movie = ["Inception", "Christopher Nolan", "2010-07-16", 148,
"Science Fiction", 8.8]
print("Title is", list_movie[0])
```

Можно определить имена для индексов:

```
TITLE = 0
DIRECTOR = 1
RELEASE_DATE = 2
DURATION = 3
GENRE = 4
RATING = 5

tuple_movie = ("Inception", "Christopher Nolan", "2010-07-16", 148,
"Science Fiction", 8.8)
print("Title is", tuple_movie[TITLE])
```

Словари выглядят немного лучше, предоставляя доступ по ключам:

```
dict_movie = {
    "title": "Inception",
    "director": "Christopher Nolan",
    "release_date": "2010-07-16",
    "duration": 148,
    "genre": "Science Fiction",
    "rating": 8.8
}
print("Title is", dict_movie["title"])
```

Именованный кортеж – это кортеж, предоставляющий доступ по индексу и ключу:

```
from collections import namedtuple

MovieNamedTuple = namedtuple("MovieNamedTuple", "title director
release_date duration genre rating")

namedtuple_movie = MovieNamedTuple("Inception", "Christopher Nolan",
"2010-07-16", 148, "Science Fiction", 8.8)

print("Title is", namedtuple_movie[0])
print("Title is", namedtuple_movie.title)
```

Нельзя написать `namedtuple_thing["name"]`. Это *tuple*, а не *dict*, поэтому индекс должен быть целым числом.

Датаклассы

Использование стандартного класса:

```
class MovieClass:
    def __init__(self,
                 title: str,
                 director: str,
                 release_date: str,
                 duration: int,
                 genre: str,
                 rating: float):

        self.title = title
        self.director = director
        self.release_date = release_date
        self.duration = duration
        self.genre = genre
        self.rating = rating
```

```
class_movie = MovieClass("Inception", "Christopher Nolan", "2010-07-
16", 148, "Science Fiction", 8.8)

print("Title is", class_movie.title)
```

Недавно в Python появился класс для хранения данных – датакласс (dataclass). Пример:

```
from dataclasses import dataclass

@dataclass
class MovieDataClass:
    title: str
    director: str
    release_date: str
    duration: int
    genre: str
    rating: float

dataclass_movie = MovieDataClass("Inception", "Christopher Nolan",
"2010-07-16", 148, "Science Fiction", 8.8)

print("Title is", dataclass_movie.title)
```

Задачи валидации

При работе с данными, обычно словарями, нужно проверять абсолютно все:

– Ключ обязателен?

– Если ключ отсутствует, есть ли значение по умолчанию?

– Существует ли ключ?

– Если да, то относится ли значение ключа к правильному типу?

– Если да, то находится ли значение в нужном диапазоне или соответствует ли оно шаблону?

Три решения отвечают хотя бы некоторым из этих требований:

– Dataclasse – часть стандартного языка Python.

– attrs – сторонний пакет, но содержит супернабор классов данных.

– Pydantic – тоже сторонний продукт, но интегрированный в FastAPI.

Создадим временную базу данных внутри приложения, а также два маршрута:

```
todo_list = []

@app.post("/todo")
async def add_todo(todo: dict) -> dict:
  todo_list.append(todo)
  return {"message": "Todo added successfully"}

@app.get("/todo")
async def retrieve_todos() -> dict:
  return {"todos": todo_list}
```

В примере POST-запрос отправляет данные в следующем формате:

```
{
  "id": id,
  "item": item
}
```

Пустой словарь может быть отправлен без возврата какой-либо ошибки. Пользователь может отправить запрос с телом, отличным от показанного ранее.

Создание модели с требуемой структурой тела запроса и назначение ее в качестве типа телу запроса гарантирует, что будут переданы только поля данных, присутствующие в модели.

Используем класс BaseModel из Pydantic:

```
from pydantic import BaseModel

class Todo(BaseModel):
  id: int
  item: str
```

Далее заменим тип переменной тела запроса с dict на Todo:

```
todo_list = []

@todo_router.post("/todo")
async def add_todo(todo: Todo) -> dict:
  todo_list.append(todo)
  return {"message": "Todo added successfully"}

@todo_router.get("/todo")
async def retrieve_todos() -> dict:
  return {"todos": todo_list}
```

Проверим новый валидатор тела запроса, отправив пустой словарь в качестве тела запроса:

```
http -v http://127.0.0.1:8000/todo <<< '{}'
```

Получаем ответ, указывающий на отсутствие полей id и item в теле запроса:

```
{
  "detail": [
    {
      "input": {},
      "loc": ["body", "id"],
      "msg": "Field required",
      "type": "missing"
    },
    {
      "input": {},
      "loc": ["body", "item"],
      "msg": "Field required",
      "type": "missing"
    }
  ]
}
```

Вложенные модели

В Pydantic модели также могут быть вложенными, например:

```
class Item(BaseModel):
  desc: str
  status: str

class Todo(BaseModel):
  id: int
  item: Item
```

В результате задача будет представлена следующим образом:

```
{
  "id": 1,
  "item": {
    "desc": "description",
    "status": "completed"
  }
}
```

Модели ответов

Добавим новый маршрут:

```
todo_list = []

@app.get("/todo")
async def retrieve_todo() -> dict:
  return {
    "todos": todo_list
}
```

Маршрут возвращает все содержимое, хранящееся в массиве todos. Чтобы указать возвращаемую информацию, нам придется либо разделить данные для отображения, либо ввести дополнительную логику. Мы можем создать модель, содержащую поля, которые хотим вернуть, и добавить ее в определение маршрута с помощью аргумента response_model.

```
class TodoItem(BaseModel):
  item: str

class TodoItems(BaseModel):
  todos: list[TodoItem]

todo_list = []

@app.get("/todo", response_model=TodoItems)
async def retrieve_todo() -> dict:
  return {
    "todos": todo_list
}
```

Проверка значений

Некоторые ограничения могут быть наложены на валидируемое значение. Целочисленное значение (conint) или число с плавающей точкой (confloat):

- gt – больше чем;
- lt – меньше чем;
- ge – больше или равно;
- le – меньше или равно;
- multiple_of – целое число, кратное значению.

Строковое (constr) значение:

– min_length – минимальная длина в символах (не в байтах);

– max_length – максимальная длина в символах;

– to_upper – преобразование в прописные буквы;

– to_lower – преобразование в строчные буквы;

– regex – сопоставление с регулярным выражением Python.

Кортеж, список или множество:

– min_items – минимальное количество элементов;

– max_items – максимальное количество элементов.

Пример:

```
from pydantic import BaseModel, constr, conint, confloat

class Movie(BaseModel):
    title: constr(min_length=1)
    director: str
    release_date: str
    duration: conint(gt=0)  # Продолжительность должна быть 
положительным целым числом
    genre: str
    rating: confloat(ge=0, le=10)  # Рейтинг должен быть в диапазоне 
от 0 до 10


bad_movie = Movie(
        duration=-120,  # Некорректная продолжительность
        rating=15.0  # Некорректный рейтинг
    )

Traceback (most recent call last):
...
duration
  Input should be greater than 0 [type=greater_than, input_value=-
120, input_type=int]
...
rating
  Input should be less than or equal to 10 [type=less_than_equal, 
input_value=15.0, input_type=float]
```

Альтернативный вариант – спецификация `Field` из библиотеки Pydantic:

```
from pydantic import BaseModel, Field

class Movie(BaseModel):
    title: str = Field(..., min_length=1, description="Title
of the movie")
    director: str = Field(..., description="Director of the movie")
    release_date: str = Field(..., description="Release date of the
movie in YYYY-MM-DD format")
    duration: int = Field(..., gt=0, description="Duration
of the movie in minutes, must be a positive integer")
    genre: str = Field(..., description="Genre of the movie")
    rating: float = Field(..., ge=0, le=10, description="Rating
of the movie, must be between 0 and 10")
```

Аргумент ... в Field() означает, что значение обязательное и значения по умолчанию не предусмотрено.

Обработка ошибок

Запросы могут возвращать ответы, сообщающие об ошибках, причем эти ответы могут не содержать достаточно информации о причине сбоя. Ошибки могут быть вызваны попыткой доступа к несуществующим ресурсам, защищенным страницам без достаточных прав доступа и даже ошибками сервера.

Ошибки в FastAPI обрабатываются путем создания исключения с помощью класса HTTPException. Класс HTTPException принимает три аргумента:

– status_code: код состояния, который будет возвращен для данного сбоя.

– detail: сопроводительное сообщение, которое будет отправлено клиенту.

– headers: необязательный параметр для ответов, требующих заголовков.

Мы можем объявить код статуса HTTP, чтобы отменить стандартный код статуса для успешных операций POST, добавив аргумент status_code в функцию декоратора:

```
@app.post("/todo", status_code=201)
async def add_todo(todo: Todo) -> dict:
  todo_list.append(todo)
    return {
        "message": "Todo added successfully."
    }
```

Пример с HTTPException:

```
from fastapi import Path, HTTPException, status

@app.get("/todo/{todo_id}")
async def get_single_todo(todo_id: int = Path(..., title="The ID
of the todo.")) -> dict:
  for todo in todo_list:
    if todo.id == todo_id:
            return {
                "todo": todo
            }
    raise HTTPException(
        status_code=status.HTTP_404_NOT_FOUND,
        detail="Todo with supplied ID doesn't exist",
        )
```

ЗАВИСИМОСТИ В FastAPI

Одной из особенностей дизайна FastAPI является техника, называемая внедрением зависимостей (Dependency Injection). Зависимость – это конкретная сущность (функция, класс), требующаяся в определенный момент. Обычный способ получить эту информацию – написать код, доставляющий ее именно тогда, когда она необходима.

В FastAPI зависимость – это то, что можно выполнять, поэтому объект зависимости должен относиться к типу Callable, включающему функции и классы – то, что можно вызвать с помощью скобок и необязательных аргументов.

Пример:

```
from fastapi import FastAPI, Depends, Body

app = FastAPI()

# функция зависимости
def user_dep(name: str = Body(embed=True), password: str =
Body(emded=True)):
  return {"name": name, "valid": True}

@app.post("/user")
def get_user(user: dict = Depends(user_dep)) -> dict:
  return user
```

В функции пути get_user() определено, что она ожидает переменную user и эта переменная получит свое значение из функции зависимости user_dep().

В аргументах функции get_user() нельзя написать user = user_dep, потому что user_dep – это объект функции.

И нельзя написать user = user_dep(), потому что это вызвало бы функцию user_dep(), когда функция get_user() была определена, а не когда она используется.

Поэтому нам нужна дополнительная вспомогательная функция FastAPI Depends(), чтобы вызывать user_dep() именно тогда, когда это необходимо.

Синтаксис в общем виде:

```
def pathfunc(name: depfunc = Depends(depfunc)):
```

Можно записать так:

```
def pathfunc(name: depfunc = Depends()):
```

name – значение (значения), возвращаемое depfunc.

Если функция зависимости проверяет что-то и не возвращает никаких значений, можно определить зависимость в декораторе пути:

```
@app.method(url, dependencies=[Depends(depfunc)])

# функция зависимости
def user_dep(name: str = Body(embed=True), password: str =
Body(emded=True)):
   if not name:
        raise

@app.post("/user", dependencies=[Depends(user_dep)])
def get_user() -> bool:
  return True
```

Далее мы подробно рассмотрим, как структурировать более крупное приложение FastAPI, включая определение нескольких объектов маршрутизатора в приложении верхнего уровня.

А пока пример, иллюстрирующий концепцию:

```
from fastapi import Depends, APIRouter

router = APIRouter(..., dependencies=[Depends(depfunc)])
```

Код приведет к вызову функции depfunc() для всех функций пути ниже объекта router.

При определении объекта приложения FastAPI верхнего уровня можно добавить к нему зависимости, применяемые ко всем его функциям пути.

```
from fastapi import FastAPI, Depends

def depfunc1():
  pass
def depfunc2():
  pass

app = FastAPI(dependencies=[Depends(depfunc1), Depends(depfunc2)])

@app.get("/main")
def get_main():
  pass
```

ШАБЛОНИЗАЦИЯ

Jinja – это шаблонизатор, написанный на Python и предназначенный для помощи в процессе рендеринга ответов API. В каждом языке шаблонизации есть переменные, которые заменяются на фактические значения, переданные им при рендеринге шаблона, и есть теги, которые управляют логикой шаблона.

В шаблонизаторе Jinja используются фигурные скобки { }, чтобы отличать выражения от обычного HTML, текста и любых других переменных в файле шаблона.

Синтаксис {{ }} называется блоком переменных.

Синтаксис {% %} содержит управляющие структуры, такие как if/else, циклы и макросы.

В языке шаблонов Jinja используются следующие три распространенных синтаксических блока:

– {% ... %} – используется для управляющих структур.

– {{ todo.item }} – используется для вывода значений выражений.

– {# Это отличный материал по FastAPI! #} – используется при написании комментариев и не отображается на странице.

Фильтры

Несмотря на схожесть синтаксиса Python и Jinja, такие модификации, как объединение строк, изменение регистра и так далее, не могут быть выполнены с помощью Python в Jinja.

Для выполнения таких модификаций в Jinja есть фильтры.

Фильтр отделяется от переменной символом пайпа (|) и может содержать необязательные аргументы в круглых скобках:

```
{{ variable | filter_name(*args) }}
```

Если аргументов нет, то определение выглядит следующим образом:

```
{{ variable | filter_name }}
```

Переменная фильтра default используется для замены переданного значения, если оно оказывается None:

```
{{ todo.item | default('This is a default todo item') }}
> This is a default todo item
```

Фильтр escape используется для вывода необработанного (сырого) HTML:

```
{{ "<title>Todo Application</title>" | escape }}
> <title>Todo Application</title>
```

Фильтры int и float используются для преобразования одного типа данных в другой:

```
{{ 3.142 | int }}
> 3
{{ 31 | float }}
> 31.0
```

Фильтр join используется для объединения элементов списка в строку, как в Python:

```
{{ ['Packt', 'produces', 'great', 'books!'] | join(' ') }}
> Packt produces great books!
```

Фильтр length используется для возврата длины переданного объекта. Он выполняет ту же роль, что и len() в Python:

```
Todo count: {{ todos | length }}
> Todo count: 4
```

Использование операторов if в Jinja аналогично их использованию в Python, операторы if используются в управляющих блоках {% %}:

```
{% if todo | length < 5 %}
    You don't have much items on your todo list!
{% else %}
    You have a busy day it seems!
{% endif %}
```

В Jinja мы также можем выполнять итерации по коллекции:

```
{% for todo in todos %}
         <b> {{ todo.item }} </b>
{% endfor %}
```

Внутри цикла for можно обращаться к специальным переменным, таким как loop.index, которая указывает индекс текущей итерации.

В документации приведен список всех специальных переменных и их описание.

Макросы

Макрос в Jinja – это функция, возвращающая HTML-строку. Основная цель использования макросов – избежать повторения кода и вместо этого использовать один вызов функции.

Например, макрос для создания поля ввода:

```
{% macro input(name, value='', type='text', size=20 %}
    <div class="form">
        <input type="{{ type }}" name="{{ name }}"
         value="{{ value|escape }}" size="{{ size }}">
    </div>
{% endmacro %}
```

Теперь, чтобы быстро создать поле ввода в форме, вызывается макрос:

```
{{ input('item') }}
```

В результате получим следующее:

```
<div class="form">
    <input type="text" name="item" value="" size="20">
</div>
```

Самая мощная особенность Jinja – наследование шаблонов. Эта возможность способствует реализации приниципа Don’t Repeat Yourself (DRY) и очень полезна в больших веб-приложениях.

Наследование шаблонов – это ситуация, когда определяется базовый шаблон, а дочерние шаблоны могут взаимодействовать, наследовать и заменять определенные секции базового шаблона.

Сначала установим пакет Jinja:

```
poetry add jinja2
```

Затем обновим маршрут GET:

```
from fastapi.templating import Jinja2Templates

@app.get("/todo", response_model=TodoItems)
async def retrieve_todo(request: Request):
    return templates.TemplateResponse("todo.html",
{         "request": request,
         "todos": todo_list
})
```

Создадим шаблон todo.html:

```
<html>
  <head>
  </head>
   <body>
    {% for todo in todos %}
      <li class="list-group-item">
      {{ loop.index }}. <a href= "/todo/{{loop.index}}">
{{ todo.item }} </a>
      </li>
    {% endfor %}
  </body>
</html>
```

АРХИТЕКТУРА ПРИЛОЖЕНИЯ

Для примера создадим планировщик событий. Структура приложения будет выглядеть следующим образом:

```
src/
  main.py
  database/
    __init__.py
    base.py
    connection.py
  routes/
    __init__.py
    events.py
    users.py
  models/
    __init__.py
    events.py
    users.py
```

Каталог routes:

– events.py: будет обрабатывать маршруты для создания, обновления и удаления событий.

– users.py: будет обрабатывать маршруты для регистрации и входа пользователей.

Каталог models:

– events.py: будет содержать определение моделей для операций с событиями.

– users.py: будет содержать определение моделей для операций с пользователями.

Реализация моделей

Определим модель Event в файле models/events.py:

```
from pydantic import BaseModel
from datetime import datetime

class Event(BaseModel):
    id: int
```

```
    title: str
    date: datetime
    description: str
    tags: list[str]
    location: str
```

Определим необязательный подкласс Config в классе Event для Swagger UI:

```
class Config:
      schema_extra = {
          "example": {
              "title": "Event title",
              "date": "2024-08-28T14:38:04",
              "description": "Event description",
              "tags": ["tag1", "tag2"],
              "location": "online"
            }
          }
```

Определим модель User в файле models/users.py:

```
from pydantic import BaseModel, EmailStr

class User(BaseModel):
    email: EmailStr
    password: str
```

Для валидации адреса почты необходим пакет pydantic[email]:

```
poetry add "pydantic[email]"
```

Создадим пример, показывающий, как хранятся и устанавливаются данные User:

```
class Config:
        schema_extra = {
            "example": {
                "email": "name@server.com",
                "password": "secret"
            }
        }
```

Реализация маршрутов User

Начнем с определения основного маршрута регистрации в файле routes/users.py:

```
from fastapi import APIRouter, HTTPException, status
from models.users import User

user_router = APIRouter(
    tags=["User"],
)

users = {}

@user_router.post("/signup")
async def sign_user_up(data: User) -> dict:
    if data.email in users:
        raise HTTPException(
            status_code=status.HTTP_409_CONFLICT,
            detail="User with supplied username exists"
        )
    users[data.email] = data
    return {
        "message": "User successfully registered!"
    }
```

В маршруте регистрации мы используем базу данных внутри приложения. Реализуем маршрут входа в систему:

```
@user_router.post("/signin")
async def sign_user_in(user: User) -> dict:
    if user.email not in users:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND,
            detail="User does not exist"
        )

    if users[user.email].password != user.password:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN,
            detail="Wrong credential passed"
        )
    return {
        "message": "User signed in successfully"
    }
```

Первым делом проверяется, существует ли пользователь в базе, и если он не существует, то возбуждается исключение. Если пользователь существует, приложение переходит к проверке совпадения паролей, после чего возвращает сообщение об успехе или исключение.

Теперь, когда мы определили маршруты для пользовательских операций, зарегистрируем их в файле main.py и запустим приложение.

```
from fastapi import FastAPI
from routes.users import user_router

import uvicorn

app = FastAPI()

# Регистрация маршрутов
app.include_router(user_router,  prefix="/user")

if __name__ == '__main__':
    uvicorn.run("main:app", host="0.0.0.0", port=8000, reload=True)
```

Протестируем точку доступа user/signup:

```
http http://127.0.0.1:8000/user/signup email="user1@server.com"
password="Str0ngPassw0rd"

HTTP/1.1 200 OK
content-length: 43
content-type: application/json
date: Tue, 13 Aug 2024 15:14:03 GMT
server: uvicorn
{
    "message": "User successfully registered!"
}
```

Протестируем точку доступа `user/signin`:

```
http http://127.0.0.1:8000/user/signin email="user1@server.com" pass-
word="Str0ngPassw0rd"

HTTP/1.1 200 OK
content-length: 41
content-type: application/json
date: Tue, 13 Aug 2024 15:15:59 GMT
server: uvicorn
{
    "message": "User signed in successfully"
}
```

Реализация маршрутов Event

Начнем с импорта зависимостей и определения маршрутизатора событий:

```
from fastapi import APIRouter, Body, HTTPException, status
from models.events import Event

event_router = APIRouter(
    tags=["Events"]
)

events = []
```

Следующим шагом будет определение маршрута для получения всех событий и событий, соответствующих заданному идентификатору:

```
@event_router.get("/", response_model=list[Event])
async def retrieve_all_events() -> list[Event]:
    return events

@event_router.get("/{id}", response_model=Event)
async def retrieve_event(id: int) -> Event:
    for event in events:
        if event.id == id:
            return event
    raise HTTPException(
        status_code=status.HTTP_404_NOT_FOUND,
        detail="Event with supplied ID does not exist"
    )
```

В маршруте мы вызываем исключение HTTP_404_NOT_FOUND, когда событие с указанным идентификатором не существует. Реализуем маршруты для создания события, удаления одного событиях:

```
@event_router.post("/new")
async def create_event(body: Event = Body(...)) -> dict:
    events.append(body)
    return {
        "message": "Event created successfully"
    }

@event_router.delete("/delete/{id}")
async def delete_event(id: int) -> dict:
    for event in events:
        if event.id == id:
            events.remove(event)
```

```
            return {
                "message": "Event deleted successfully"
            }

    raise HTTPException(
        status_code=status.HTTP_404_NOT_FOUND,
        detail="Event with supplied ID does not exist"
    )
```

Теперь, когда мы реализовали маршруты, обновим конфигурацию маршрутов, чтобы включить маршруты событий в файл main.py:

```
from fastapi import FastAPI
from routes.user import user_router
from routes.events import event_router
import uvicorn

app = FastAPI()

# Регистрация маршрутов
app.include_router(user_router, prefix="/user")
app.include_router(event_router, prefix="/event")

if __name__ == "__main__":
    uvicorn.run("main:app", host="0.0.0.0", port=8080,
    reload=True)
```

Протестируем точку доступа event/:

```
http http://127.0.0.1:8000/event/

HTTP/1.1 200 OK
content-length: 2
content-type: application/json
date: Tue, 13 Aug 2024 15:25:38 GMT
server: uvicorn

[]
```

Протестируем точку доступа event/new:

```
http http://127.0.0.1:8000/event/new id=1 title="Event title"
date="2024-08-28T14:38:04" description="Event description"
tags:='["tag1", "tag2"]' location="online"

HTTP/1.1 200 OK
content-length: 40
content-type: application/json
date: Tue, 13 Aug 2024 15:30:53 GMT
server: uvicorn
```

```
{
    "message": "Event created successfully"
}
```

Протестируем точку доступа event/{id} (GET):

```
http http://127.0.0.1:8000/event/1

HTTP/1.1 200 OK
content-length: 286
content-type: application/json
date: Tue, 13 Aug 2024 15:35:46 GMT
server: uvicorn

{
    "description": "Event description",
    "id": 1,
    "date": "2024-08-28T14:38:04",
    "location": "online",
    "tags": [
        "tag1",
        "tag2"
    ],
    "title": "Event title"
}
```

Протестируем точку доступа event/{id} (метод DELETE):

```
http DELETE http://127.0.0.1:8000/event/delete/1

HTTP/1.1 200 OK
content-length: 40
content-type: application/json
date: Tue, 13 Aug 2024 15:37:35 GMT
server: uvicorn

{
    "message": "Event deleted successfully"
}
```

При повторном запросе получим ответ:

```
HTTP/1.1 404 Not Found
content-length: 50
content-type: application/json
date: Tue, 13 Aug 2024 15:38:27 GMT
server: uvicorn

{
    "detail": "Event with supplied ID does not exist"
}
```

ПОДКЛЮЧЕНИЕ БАЗЫ ДАННЫХ

Первым шагом для интеграции с базой данных SQL является установка библиотеки SQLAlchemy.

```
poetry add sqlalchemy
poetry add asyncpg
```

В SQLAlchemy подключение к базе данных осуществляется с помощью движка. Движок создается функцией create_async_engine(), импортируемой из модуля sqlalchemy.ext.asyncio. Функция create_async_engine() принимает в качестве аргумента URL базы данных.

URL базы данных имеет вид:

```
postgresql+asyncpg://username:password@server:5432/dbname.
```

Также метод принимает необязательный аргумент echo, который при установке значения True выводит команды SQL при выполнении операций.

Определим конфигурацию, необходимую для создания базы данных и таблицы, в файле connection.py:

```
from sqlalchemy.ext.asyncio import create_async_engine

DATABASE_URL = "post-
gresql+asyncpg://username:password@127.0.0.1:5432/app_db"
engine = create_async_engine(DATABASE_URL, echo=True, future=True)
```

В приложении будет много участков кода, в которых нужно использовать сессии, и может быть неудобно постоянно передавать туда движок и другие параметры.

Функция фабрики async_sessionmaker из sqlalchemy.ext.asyncio позволяет создать собственный класс AsyncSession, в котором будут учтены все параметры:

```
session_maker = async_sessionmaker(engine, class_=AsyncSession,
expire_on_commit=False)
```

Добавим контекстный менеджер, который будет создавать асинхронную сессию для работы с базой данных.

```
async def get_session() -> AsyncGenerator[AsyncSession, None]:
    async with session_maker() as session:
        yield session
```

После завершения работы с сессией контекстный менеджер автоматически закрывает ее, обеспечивая корректное освобождение ресурсов.

Создадим родительский класс, унаследованный от декларативного базового класса, для моделей SQLAlchemy в файле base.py:

```
from sqlalchemy.orm import DeclarativeBase

class Base(DeclarativeBase):
    pass
```

Создадим модель SQLAlchemy для событий в файле models/events.py:

```
from database.base import Base
from sqlalchemy import ARRAY, String, DateTime
from sqlalchemy.orm import Mapped, mapped_column
from datetime import datetime

class Event(Base):
    __tablename__ = "events"

    id: Mapped[int] = mapped_column(primary_key=True)
    title: Mapped[str] = mapped_column(String(64))
    date: Mapped[datetime] = mapped_column(DateTime)
    description: Mapped[str] = mapped_column(String(256))
    tags: Mapped[list[str]] = mapped_column(ARRAY(String(16)))
    location: Mapped[str] = mapped_column(String(64))
```

Обновим классы моделей Pydantic для запросов на получение событий и их создание:

```
class EventRequest(BaseModel):
    title: str
    date: datetime
    description: str
    tags: list[str]
    location: str

class EventCreate(EventRequest):
    pass
```

Добавим еще один класс Pydantic, который будет использоваться при операциях обновления:

```
class EventUpdate(EventCreate):
    title: str | None = None
    date: datetime | None = None
    description: str | None = None
    tags: list[str] | None = None
    location: str | None = None
```

Определим конфигурацию, необходимую для таблиц, в файле connection.py:

```
from database.base import Base
# ...
async def init_db():
    async with engine.begin() as conn:
        # await conn.run_sync(Base.metadata.drop_all)
        await conn.run_sync(Base.metadata.create_all)
```

Обновим файл main.py, добавив в него следующий код:

```
from fastapi import FastAPI
from fastapi.responses import RedirectResponse
from database.connection import conn

from routes.users import user_router
from routes.events import event_router

import uvicorn

app = FastAPI()

app.include_router(user_router,  prefix="/user")
app.include_router(event_router, prefix="/event")

@app.on_event("startup")
async def on_startup():
    await init_db()

@app.get("/")
async def home():
    return RedirectResponse(url="/event/")

if __name__ == '__main__':
    uvicorn.run("main:app", host="0.0.0.0", port=8000, reload=True)
```

В файле routes/events.py обновим импорт для включения классов моделей событий, а также функции get_session():

```
from fastapi import APIRouter, Depends, HTTPException, status
from sqlalchemy.ext.asyncio.session import AsyncSession
from sqlalchemy import select
from database.connection import get_session
from models.events import EventRequest, EventCreate, EventUpdate,
Event
```

Функция get_session() импортируется для того, чтобы маршруты могли получить доступ к созданному объекту сессии.

Далее обновим функцию POST-маршрута, отвечающую за создание нового события, create_event():

```
@event_router.post("/new")
async def create_event(
    new_event: EventCreate, session: AsyncSession = De-
pends(get_session)) -> dict:
    event = Event(**new_event.model_dump()) # распаковки словаря
в аргументы конструктора
    session.add(event)
    await session.commit()
    await session.refresh(event)
    return {"message": "Event created successfully"}
```

Обновим маршрут GET, который запрашивает список событий, для получения данных из базы данных:

```
@event_router.get("/", response_model=list[EventRequest])
async def retrieve_all_events(
    session: AsyncSession = Depends(get_session),) -> list[Event]:
    statement = select(Event)
    result = await session.execute(statement)
    events = result.scalars().all()
    return list(events)
```

Аналогичным образом обновим маршрут отображения данных события при получении его идентификатора:

```
@event_router.get("/{id}", response_model=EventRequest)
async def retrieve_event(
    id: int, session: AsyncSession = Depends(get_session)) -> Event |
dict:
    event = await session.get(Event, id)
    if event:
        return event
```

```
    raise HTTPException(
        status_code=status.HTTP_404_NOT_FOUND,
        detail="Event with supplied ID does not exist",
    )
```

Добавим маршрут PATCH в файл routes/events.py:

```
@event_router.patch("/edit/{id}", response_model=EventCreate)
async def update_event(id: int, new_data: EventUpdate, session:
AsyncSession = Depends(get_session)) -> Event | dict:
```

В теле функции добавим следующий блок кода для получения существующего события и обработки его изменений:

```
event = await session.get(Event, id)
  if event:
      event_data = new_data.model_dump(exclude_unset=True) #
исключение полей, которые не были установлены явно
      for key, value in event_data.items():
          setattr(event, key, value)
      session.add(event)
      await session.commit()
      await session.refresh(event)
      return event
  raise HTTPException(
      status_code=status.HTTP_404_NOT_FOUND,
      detail="Event with supplied ID does not exist",
  )
```

В файле events.py обновим маршрут удаления, определенный ранее:

```
@event_router.delete("/delete/{id}")
async def delete_event(id: int, session: AsyncSession =
Depends(get_session)) -> dict:
    event = await session.get(Event, id)
    if event:
        await session.delete(event)
        await session.commit()
        return {"message": "Event deleted successfully"}
    raise HTTPException(
        status_code=status.HTTP_404_NOT_FOUND,
        detail="Event with supplied ID does not exist",
    )
```

В этом блоке кода функция проверяет, существует ли событие, идентификатор которого был указан, а затем удаляет его из базы данных.

АУТЕНТИФИКАЦИЯ

FastAPI поддерживает несколько методов аутентификации: базовую HTTP-аутентификацию, основанную на cookie и аутентификацию с помощью Bearer-токена.

Базовая HTTP-аутентификация (Basic HTTP): в этом методе аутентификации учетные данные пользователя, которые обычно представляют собой имя пользователя и пароль, отправляются в HTTP-заголовке Authorization. В ответе возвращается заголовок WWW-Authenticate, содержащий значение Basic и необязательный параметр realm (область), который указывает ресурс, к которому сделан запрос аутентификации.

Файлы cookie: файлы cookie используются, когда данные должны храниться на стороне клиента. Приложения FastAPI также могут использовать файлы cookie для хранения пользовательских данных, которые могут быть извлечены сервером для целей аутентификации.

Аутентификация с помощью Bearer-токена (Bearer token): этот метод аутентификации подразумевает использование токенов безопасности, называемых Bearer-токенами. Эти токены отправляются вместе с ключевым словом Bearer в запросе заголовка Authorization. Наиболее часто используемый токен – JWT (JSON Web Token), который обычно представляет собой словарь, включающий идентификатор пользователя и время истечения срока действия токена.

Внедрение зависимостей

В FastAPI зависимость может быть определена как функция или класс.

Созданная зависимость дает доступ к ее базовым атрибутам или методам, устраняя необходимость создавать эти объекты в наследующих их функциях.

Внедрение зависимости помогает сократить повторение кода в некоторых случаях, например, при аутентификации и авторизации:

```
async def get_user(token: str):
    user = decode_token(token)
    return user
```

Эта зависимость является функцией, которая принимает токен в качестве аргумента и возвращает параметр user из внешней функции decode_token.

Чтобы использовать эту зависимость, объявленный аргумент зависимой функции устанавливается с параметром Depends, например:

```
from fastapi import Depends

@router.get("/user/me")
async get_user_details(user: User = Depends(get_user)):
  return user
```

OAuth2 Flow

Будем использовать схему аутентификации OAuth2, которая требует, чтобы клиент отправил имя пользователя и пароль в качестве данных формы. Имя пользователя в нашем случае – это адрес электронной почты, используемый при создании учетной записи.

Когда данные формы отправляются клиентом на сервер, в качестве ответа отправляется токен доступа (access token), который представляет собой подписанный JWT-токен. Перед созданием токена доступа для дальнейшей авторизации выполняется проверка подлинности отправленных на сервер учетных данных.

Для авторизации аутентифицированного пользователя в JWT добавляется префикс Bearer при отправке данных в заголовке для авторизации действия на сервере.

В каталоге auth создадим четыре файла:

– jwt_handler.py: этот файл будет содержать функции, необходимые для кодирования и декодирования JWT-токенов.

– authenticate.py: этот файл будет содержать зависимость `authenticate`, которая будет внедряться в маршруты для обеспечения аутентификации и авторизации.

– hash_password.py: этот файл будет содержать функции, которые будут использоваться для хеширования пароля пользователя при регистрации и сравнения паролей при входе в систему.

– __init__.py: этот файл указывает на содержимое каталога как модуля.

Хеширование паролей

Раньше мы хранили пароли пользователей в виде обычного текста. Это крайне небезопасная и запрещенная практика при создании API. Пароли должны быть зашифрованы или хешированы с помощью соответствующих библиотек.

Мы будем хешировать пароли пользователей с помощью алгоритма bcrypt. Для этого установим пакет passlib:

```
poetry add "passlib[bcrypt]"
```

Создадим функции для хеширования паролей в файле hash_password.py:

```
from passlib.context import CryptContext

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

class HashPassword:
    def create_hash(self, password: str):
        return pwd_context.hash(password)

    def verify_hash(self, plain_password: str, hashed_password: str):
        return pwd_context.verify(plain_password, hashed_password)
```

```
from passlib.context import CryptContext

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

class HashPassword:
```

```
    def create_hash(self, password: str):
        return pwd_context.hash(password)

    def verify_hash(self, plain_password: str, hashed_password: str):
        return pwd_context.verify(plain_password, hashed_password)
```

В предыдущем блоке кода мы начинаем с импорта CryptContext, который принимает схему bcrypt для хеширования переданных ему строк. Контекст хранится в переменной pwd_context, предоставляя доступ к методам, необходимым для выполнения задачи. Затем определяется класс HashPassword, который содержит два метода, create_hash() и verify_hash():

– Метод create_hash() принимает строку и возвращает хешированное значение.

– Метод verify_hash() принимает обычный пароль и хешированный пароль и сравнивает их. Функция возвращает булево значение, указывающее, совпадают ли переданные значения или нет.

Создадим модель SQLAlchemy для событий в файле models/users.py:

```
class User(Base):
    __tablename__ = "users"

    id: Mapped[int] = mapped_column(primary_key=True)
    email: Mapped[str] = mapped_column(String(32), unique=True)
    password: Mapped[str] = mapped_column(String(72))
```

Обновим классы моделей Pydantic для операций входа и регистрации:

```
class SignInUser(BaseModel):
    email: EmailStr
    password: str

    class Config:
        schema_extra = {
            "example": {
                "email": "name@server.com",
                "password": "secret",
            }
        }

class SignUpUser(SignInUser):
    pass
```

Обновим маршрут регистрации пользователей в routes/users.py, добавив хеширование пароля:

```
from models.users import SignInUser, SignUpUser, User
from auth.hash_password import HashPassword

hash_password = HashPassword()

@user_router.post("/signup")
async def sign_user_up(
    data: SignUpUser, session: AsyncSession = Depends(get_session)) -
> dict:
    statement = select(User).where(User.email == data.email)
    result = await session.execute(statement)
    user = result.scalar()
    if user:
        raise HTTPException(
            status_code=status.HTTP_409_CONFLICT,
            detail="User with supplied email exists",
        )

    user = User(**data.model_dump())
    hashed_password = hash_password.create_hash(data.password)
    user.password = hashed_password
    session.add(user)
    await session.commit()
    await session.refresh(user)

    return {"message": "User successfully registered!"}
```

Обновим маршрут входа пользователй routes/users.py, добавив проверку хешей паролей:

```
@user_router.post("/signin")
async def sign_user_in(data: SignInUser, session: AsyncSession =
Depends(get_session)) -> dict:
    statement = select(User).where(User.email == data.email)
    result = await session.execute(statement)
    user = result.scalar()
    if not user:
        raise HTTPException(
            status_code=status.HTTP_404_NOT_FOUND, detail="User does
not exist"
        )

    if not hash_password.verify_hash(data.password, user.password):
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN, detail="Wrong cre-
dential passed"
        )
    return {"message": "User signed in successfully"}
```

Для работы с токеном используем пакет pyjwt:

```
poetry add pyjwt
```

Полезная нагрузка токена будет состоять из идентификатора пользователя и его времени истечения:

```
# jwt_handler.py
import time
import jwt

SECRET_KEY: str = "SECRET_KEY"

def create_access_token(user: str) -> str:
    payload = {"user": user, "expires": time.time() + 3600}
    token = jwt.encode(payload, SECRET_KEY, algorithm="HS256")
    return token
```

Значение expires устанавливается равным часу с момента создания токена.

Затем payload передается в метод encode(), который принимает три параметра:

– payload: словарь, содержащий значения для хеширования.

– key: ключ, используемый для подписи полезной нагрузки.

– algorithm: алгоритм, используемый при подписании payload (HS256).

Создадим функцию для проверки подлинности токена, отправленного в приложение:

```
from datetime import datetime
import jwt
from jwt.exceptions import InvalidTokenError
...
def verify_access_token(token: str) -> dict:
    try:
        data = jwt.decode(token, SECRET_KEY, algorithms=["HS256"])
        expire = data.get("expires")
        if expire is None:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
                detail="No access token supplied",
            )
        if datetime.utcnow() > datetime.utcfromtimestamp(expire):
            raise HTTPException(
                status_code=status.HTTP_403_FORBIDDEN, detail="Token
```

```
expired!"
            )
        return data

    except InvalidTokenError:
        raise HTTPException(
            status_code=status.HTTP_400_BAD_REQUEST, detail="Invalid
token"
        )
```

Реализуем функцию зависимости, которая будет вводиться в маршруты событий. Эта функция будет служить единым источником истины для получения пользователя.

В файл auth/authenticate.py добавим следующий код:

```
from fastapi import Depends, HTTPException, status
from fastapi.security import OAuth2PasswordBearer
from auth.jwt_handler import verify_access_token

oauth2_scheme = OAuth2PasswordBearer(tokenUrl="/user/signin")

async def authenticate(token: str = Depends(oauth2_scheme)) -> str:
    if not token:
        raise HTTPException(
            status_code=status.HTTP_403_FORBIDDEN, detail="Sign in
for access"
        )
    decoded_token = verify_access_token(token)
    return decoded_token["user"]
```

В предыдущем блоке кода мы начинаем с импорта необходимых зависимостей:

– Depends: добавляет oauth2_scheme в функцию как зависимость.

– OAuth2PasswordBearer: класс сообщает приложению, что присутствует схема безопасности.

– verify_access_token: функция используется для проверки валидности токена.

Обновление маршрутов

В файле routes/users.py добавим импорт:

```
from fastapi.security import OAuth2PasswordRequestForm
from auth.jwt_handler import create_access_token
```

Мы импортировали класс OAuth2PasswordRequestForm из модуля fastapi.security FastAPI, который будет внедрен в маршрут входа для получения отправленных учетных данных.

Обновим функцию маршрута sign_user_in():

```
async def sign_user_in(
    data: OAuth2PasswordRequestForm = Depends(), session:
AsyncSession = Depends(get_session),
) -> TokenResponse | dict:
    statement = select(User).where(User.email == data.username)
    result = await session.execute(statement)
    ...
    access_token = create_access_token(data.username)
    return {"access_token": access_token, "token_type": "Bearer"}
```

Создадим модель ответа для маршрута login в models/users.py, чтобы заменить класс модели UserSignIn, который больше не используется:

```
class TokenResponse(BaseModel):
    access_token: str
    token_type: str
```

Обновим импорт и модель ответа для маршрута регистрации:

```
from models.users import TokenResponse, SignUpUser, User

@user_router.post("/signin", response_model=TokenResponse)
```

Внедрим зависимость аутентификации в функции маршрутов событий routes/events.py:

```
from auth.authenticate import authenticate
...
@event_router.get("/", response_model=list[EventRequest])
async def retrieve_all_events(
    user: str = Depends(authenticate),
    session: AsyncSession = Depends(get_session),
    ) -> list[Event]:
...
@event_router.get("/{id}", response_model=EventRequest)
async def retrieve_event(
    id: int,
    user: str = Depends(authenticate),
    session: AsyncSession = Depends(get_session),
    ) -> Event:
...
```

```python
@event_router.post("/new")
async def create_event(
    data: EventCreate,
    user: str = Depends(authenticate),
    session: AsyncSession = Depends(get_session),
    ) -> dict:
...

@event_router.patch("/edit/{id}", response_model=EventCreate)
async def update_event(
    id: int,
    data: EventUpdate,
    user: str = Depends(authenticate),
    session: AsyncSession = Depends(get_session),
    ) -> Event | dict:
...

@event_router.delete("/delete/{id}")
async def delete_event(
    id: int,
    user: str = Depends(authenticate),
    session: AsyncSession = Depends(get_session),
    ) -> dict:
...
```

Откроем документацию SwaggerUI по адресу http://127.0.0.1:8000/docs, чтобы убедиться, что тело запроса соответствует спецификациям OAuth2 (рис. 1).

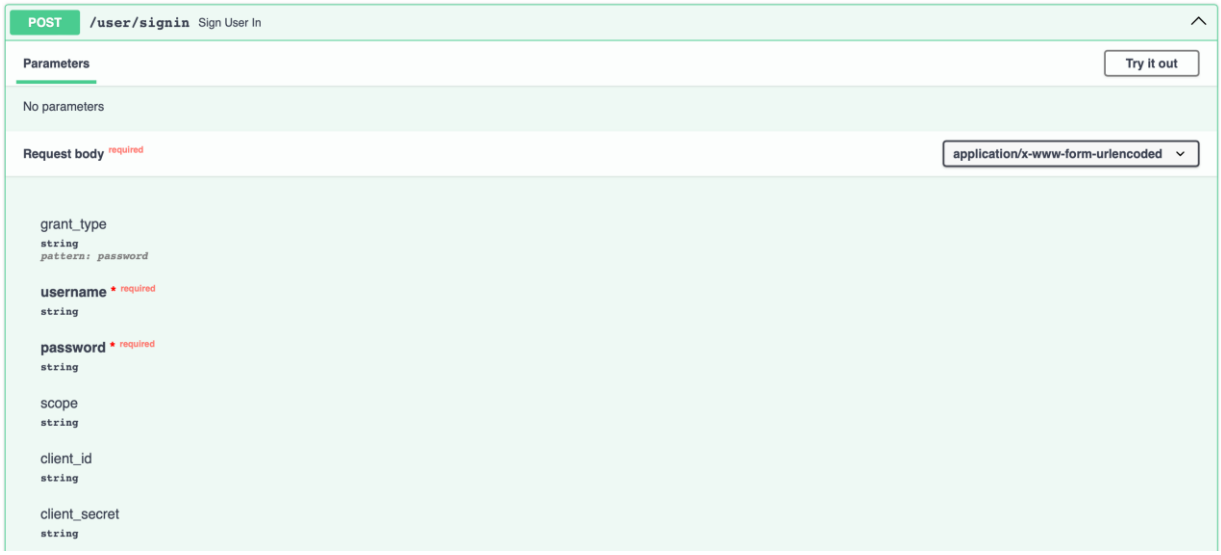

Рис. 1. Документация SwaggerUI

Протестируем маршруты:

```
http -v --form http://127.0.0.1:8000/user/signin
username="user1@server.com" password="Str0ngPassw0rd"

POST /user/signin HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Content-Length: 44
Content-Type: application/x-www-form-urlencoded; charset=utf-8
Host: 127.0.0.1:8000
User-Agent: HTTPie/3.2.2

username=user3%40server.com&password=Str0ngPassw0rd


HTTP/1.1 200 OK
content-length: 196
content-type: application/json
date: Mon, 19 Aug 2024 09:14:59 GMT
server: uvicorn

{
    "access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
    "token_type": "Bearer"
}
```

В SwaggerUI можно нажать кнопку Authorize, и отобразится модальное окно входа. Ввод учетных данных и пароля приведет к результату, показанному на рис. 2.

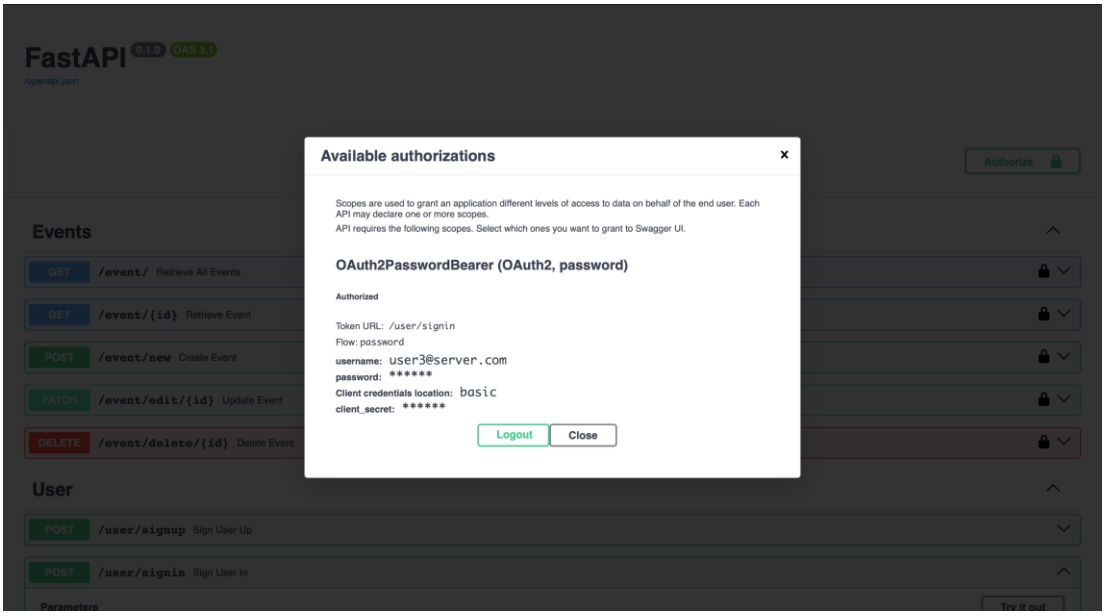

Рис. 2. Авторизация в SwaggerUI

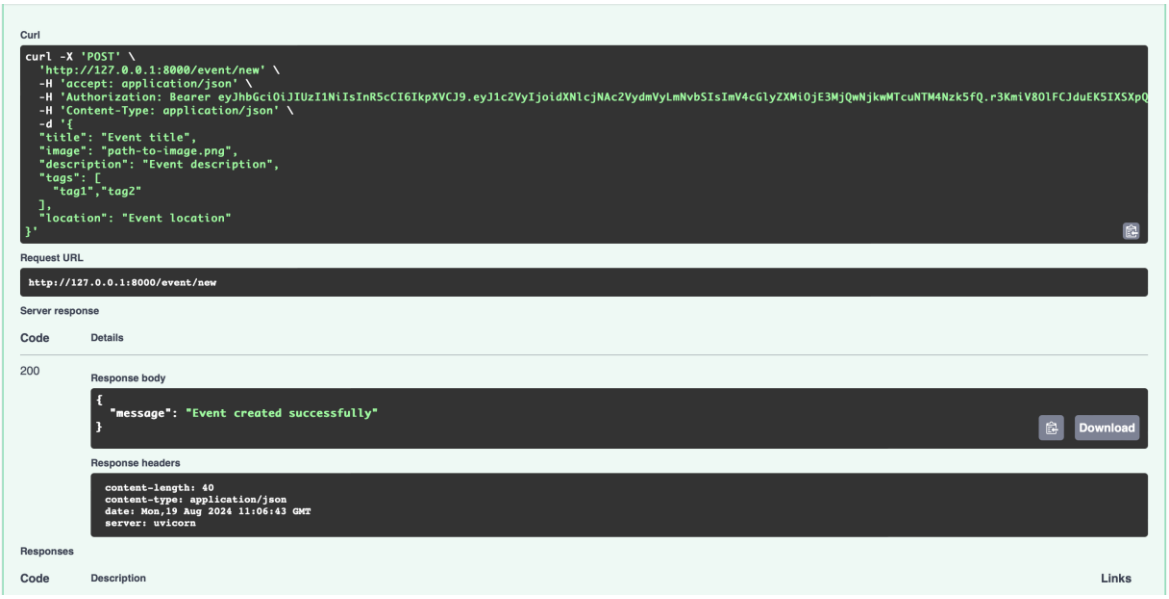

Рис. 3. Создание события в SwaggerUI

Теперь, когда мы успешно вошли в систему, мы можем создать событие (рис. 3).

Те же операции можно выполнить из командной строки:

```
http http://127.0.0.1:8000/event/new id=1 title="Event title"
image="https://link-to-image.com/image.png" description="Event
description" tags:='["tag1", "tag2"]' location="online" -A bearer -
a eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

HTTP/1.1 200 OK
content-length: 40
content-type: application/json
date: Mon, 19 Aug 2024 11:12:44 GMT
server: uvicorn

{
    "message": "Event created successfully"
}
```

Если мы попытаемся создать событие без передачи заголовка авторизации с действительным токеном, будет возвращена ошибка HTTP 401 Unauthorized:

```
http http://127.0.0.1:8000/event/new id=1 title="Event title"
date="2024-08-28T14:38:04" description="Event description"
tags:='["tag1", "tag2"]' location="online"
```

```
HTTP/1.1 401 Unauthorized
content-length: 30
content-type: application/json
date: Mon, 19 Aug 2024 11:15:02 GMT
server: uvicorn
www-authenticate: Bearer

{
    "detail": "Not authenticated"
}
```

Обновим защищенные маршруты следующим образом:

– Маршрут POST: созданное событие добавляется в список событий, принадлежащих пользователю.

– Маршрут PATH: созданное пользователем событие может быть обновлено только пользователем, его создавшим.

– Маршрут DELETE: созданное пользователем событие может быть удалено только пользователем, его создавшим.

Добавим поле создателя события (creator) в класс Event в models/events.py:

```python
class Event(Base):
    __tablename__ = "events"

    id: Mapped[int] = mapped_column(primary_key=True)
    creator: Mapped[str] = mapped_column(String(32), unique=True)
    title: Mapped[str] = mapped_column(String(64))
    date: Mapped[datetime] = mapped_column(DateTime)
    description: Mapped[str] = mapped_column(String(256))
    tags: Mapped[list[str]] = mapped_column(ARRAY(String(16)))
    location: Mapped[str] = mapped_column(String(64))
```

Изменим маршрут POST для сохранения поля создателя при создании нового события в routes/events.py:

```python
async def create_event(
    data: EventCreate,
    user: str = Depends(authenticate),
    session: AsyncSession = Depends(get_session),) -> dict:
    event = Event(**data.model_dump())
    event.creator = user
    ...
```

Если создается новое событие, то оно сохраняется с адресом электронной почты создателя:

```
http http://127.0.0.1:8000/event/new title="Event title" date="2024-
08-28T14:38:04" description="Event description" tags:='["tag1",
"tag2"]' location="Event location" -A bearer -a eyJhbGciOi-
JIUzI1NiIsInR5cCI6IkpXVCJ9...

HTTP/1.1 200 OK
content-length: 40
content-type: application/json
date: Mon, 19 Aug 2024 18:46:53 GMT
server: uvicorn

{
    "message": "Event created successfully"
}
```

Обновим маршрут PATCH:

```
async def update_event(
    id: int,
    data: EventUpdate,
    user: str = Depends(authenticate),
    session: AsyncSession = Depends(get_session),) -> Event | dict:
    event = await session.get(Event, id)
    if event:
        if event.creator != user:
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST,
detail="Operation not allowed"
            )
    ...
```

Если текущий пользователь не может редактировать событие, функция вызывает исключение HTTP 400 Bad Request:

```
http PATCH http://127.0.0.1:8000/event/edit/1 title="New event title"
-A bearer -a eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

HTTP/1.1 400 Bad Request
...
{
    "detail": "Operation not allowed"
}
```

Обновим маршрут DELETE:

```
async def delete_event(
    id: int,
    user: str = Depends(authenticate),
    session: AsyncSession = Depends(get_session),) -> dict:
    event = await session.get(Event, id)
    if event:
        if event.creator != user:
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND,
detail="Operation not allowed"
            )
  ...
```

Проверка:

```
http DELETE http://127.0.0.1:8000/event/delete/1 -A bearer -
a eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...

HTTP/1.1 400 Bad Request
...
{
    "detail": "Operation not allowed"
}
```

CORS

Промежуточное программное обеспечение

FastAPI позволяет вставлять на веб-уровень код, выполняющий:

- перехват запроса;
- операции с запросом;
- передачу запроса функции пути;
- перехват ответа, возвращаемого исполняющей функцией;
- операции с ответом;
- возврат ответа вызывающей стороне.

Это похоже на то, что декоратор в Python делает с «оборачиваемой» функцией.

В некоторых случаях можно использовать либо промежуточное ПО (middleware), либо внедрение зависимостей с помощью Depends().

Промежуточное ПО удобнее для решения более глобальных вопросов безопасности, таких как CORS.

Совместное использование ресурсов разными источниками (Cross-Origin Resource Sharing, CORS) предполагает связь между другими доверенными серверами и вашим сайтом. Если на сайте весь код фронтенда и бэкенда находится в одном месте, то проблем не возникнет. Но в наши дни часто встречается ситуация, когда фронтенд на JavaScript общается с бэкендом, написанным на чем-то вроде FastAPI. Эти серверы не будут иметь одинакового происхождения, так как у них будут отличаться следующие параметры:

– протокол – HTTP или HTTPS;

– домен – интернет-домен, например yandex.ru или 127.0.0.1;

– порт – числовой TCP/IP-порт в этом домене, например 80, 443 или 8000.

Как бэкенд может отличить надежный фронтенд от приложения злоумышленника? Это задача CORS–технологии, определяющей, чему доверяет бэкенд.

Наиболее известными способами ограничить достук к бэкенду являются:

– заголовки запросов Origin;

– HTTP-методы;

– HTTP-заголовки;

– тайм-аут кэша CORS.

В примере показано, как разрешить только один фронтенд-сервер (с доменом https://site.com), а также любые HTTP-заголовки и методы:

```
from fastapi.middleware.cors import CORSMiddleware

app = FastAPI()

app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://site.com",],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
```

Более подробную информацию о CORS можно найти в документации FastAPI – https://fastapi.tiangolo.com/tutorial/cors/.

МИГРАЦИИ

Alembic – инструмент миграции базы данных, который является частью семейства SQLAlchemy.

Установка пакета:

```
poetry add alembic
```

Первым шагом для включения миграции баз данных является создание репозитория миграции с помощью команды alembic init:

```
alembic init -t async migrations
```

Аргумент migrations – это имя подкаталога, который создается в каталоге проекта, где будут храниться скрипты миграции базы данных.

В дополнение к этому подкаталогу в каталоге проекта создается файл alembic.ini.

Для проектов, в которых используется система контроля версий (Git), файл alembic.ini и все содержимое подкаталога migrations следует рассматривать как исходный код и поддерживать вместе с исходными файлами приложения.

Файлы, созданные командой alembic init, изначально не имеют информации о том, какую базу данных использует проект.

Чтобы указать Alembic на базу данных проекта, необходимо внести несколько простых изменений в конфигурацию.

Единого способа сделать это не существует, но для проекта наиболее удобным вариантом будет редактирование файла env.py, расположенного внутри migrations.

В верхней части файла импортируем класс MetaData из SQLAlchemy, engine и модели:

```
from src.database.connection import engine
from src.models.events import Event
from src.models.users import User
```

Затем надо найти строку:

```
target_metadata = None
```

и объединить метаданные наших моделей:

```
target_metadata = Event.metadata
target_metadata = User.metadata
```

и добавить еще одну строку:

```
config.set_main_option("sqlalchemy.url", en-
gine.url.render_as_string(hide_password=False))
```

В значении переменной target_metadata Alembic ожидает экземпляр метаданных, используемый приложением.

Последняя строка вставляет значение для параметра sqlalchemy.url в объект конфигурации Alembic.

Поскольку приложение создает экземпляр движка с этим URL, наиболее удобным (хотя, возможно, не самым эффективным) является получение URL из этого объекта.

Поскольку миграции Alembic находятся на одном уровне с каталогом проекта src, а в модулях проекта используются абсолютные импорты, добавим в начало файла env.py путь к каталогу:

```
import sys
sys.path.append('./src')
```

Еще одна дополнительная настройка, которую рекомендуется выполнить, – это задание формата именования файлов с миграциями в alembic.ini:

```
file_template = %%(year)d-%%(month).2d-%%(day).2d_%%(slug)s
```

Для генерации начальной миграции необходимо, чтобы база данных была полностью пустой, поскольку это заставит Alembic сгенерировать миграцию, которая создаст таблицы:

```
alembic revision --autogenerate -m "add users, events"
```

В результате создания миграции в каталоге migrations появится файл с кодом начальной миграции:

```
def upgrade() -> None:
    op.create_table('events',
    sa.Column('id', sa.Integer(), nullable=False),
    sa.Column('creator', sa.String(length=32), nullable=False),
    ...
    )
    op.create_table('users',
    sa.Column('id', sa.Integer(), nullable=False),
    sa.Column('email', sa.String(length=32), nullable=False),
    ...
    )
```

Для обновления базы данных необходимо выполнить команду:

```
alembic upgrade head
```

БИБЛИОТЕКА Pytest

Тестирование является неотъемлемой частью цикла разработки приложений. Тестирование приложений проводится для того, чтобы обеспечить правильное функционирование приложения и легко обнаружить аномалии в приложении перед его развертыванием в производственной среде.

Модульное тестирование (Unit testing) – это процедура тестирования, при которой проверяются отдельные компоненты приложения. Такая форма тестирования позволяет проверить работоспособность отдельных компонентов. Например, модульные тесты используются при тестировании отдельных маршрутов в приложении, чтобы убедиться в правильности возвращаемых ответов.

Мы будем использовать Pytest – библиотеку тестирования Python. Хотя Python поставляется со встроенной библиотекой модульного тести-

рования unittest, библиотека Pytest имеет более простой синтаксис и более предпочтительна для тестирования приложений.

Установка:

```
poertry add -G dev pytest
```

Ключевые особенности Pytest

Обнаружение тестов – тест для файла Python с префиксом test_ или суффиксом *_test* в имени будет запущен автоматически. Это сопоставление имен файлов переходит в подкаталоги, выполняя столько тестов, сколько в них содержится.

Конструкция assert – оператор выявления ошибок assert выводит то, что ожидалось, и то, что произошло на самом деле.

Фикстуры – эти функции могут запускаться один раз для всего тестового скрипта или выполняться для каждого теста (его области видимости), предоставляя тестовым функциям такие параметры, как стандартные тестовые данные или инициализация базы данных.

Фикстуры – это своего рода внедрение зависимости, подобное предлагаемому FastAPI для функций пути веб-приложения, – конкретные данные передаются общей тестовой функции.

Параметризация – обеспечивает несколько тестовых данных для тестовой функции.

Создадим каталог tests, в котором будут храниться тестовые файлы приложения.

Для тестирования асинхронных API мы воспользуемся HTTPx и установим библиотеку pytest-asyncio, которая позволит нам тестировать асинхронные API:

```
poertry add -G dev httpx pytest-asyncio
```

Создадим конфигурационный файл pytest.ini и добавим в него следующий код:

```
[pytest]
asyncio_mode = auto
pythonpath = src
addopts = -p no:warnings
```

Конфигурационный файл считывается при запуске pytest. Создадим файл conftest.py, который будет отвечать за создание экземпляра приложения в каталоге tests, и выполним импорт необходимых зависимостей:

```
import asyncio
import pytest_asyncio
from httpx import AsyncClient, ASGITransport

from sqlalchemy.ext.asyncio import create_async_engine, AsyncEngine,
AsyncSession
from sqlalchemy.pool import NullPool

from main import app
from database.base import Base
from database.connection import get_session, session_maker
```

Начнем с создания асинхронного движка SQLAlchemy:

```
DATABASE_URL = "post-
gresql+asyncpg://username:password@127.0.0.1:5432/test_db"

engine: AsyncEngine = create_async_engine(DATABASE_URL,
poolclass=NullPool, echo=True)
```

Создадим контекстный менеджер для переопределения сессии базы данных:

```
@asynccontextmanager
   async def override_get_session():
       async with AsyncSession(engine) as session:
           yield session
```

Этот контекстный менеджер используется для создания сессии базы данных, которая будет использоваться в тестах.

Создадим функцию для инициализации базы данных:

```
async def init_db():
    async with engine.begin() as conn:
        await conn.run_sync(Base.metadata.drop_all)
        await conn.run_sync(Base.metadata.create_all)
```

Функция init_db отвечает за инициализацию базы данных: удаление всех таблиц и создание их заново.

Создадим фикстуру для инициализации базы данных перед запуском тестов:

```
@pytest_asyncio.fixture(scope="session", autouse=True)
   async def initialize_database():
       await init_db()
       yield
```

Эта фикстура инициализирует базу данных один раз перед запуском всех тестов.

Создадим фикстуру для создания и закрытия событийного цикла:

```
@pytest_asyncio.fixture(scope="function")
   def event_loop():
       policy = asyncio.get_event_loop_policy()
       loop = policy.new_event_loop()
       yield loop
       loop.close()
```

Эта фикстура создает новый событийный цикл для каждого теста и закрывает его после завершения теста.

Добавим фикстуру для создания сессии базы данных для каждого теста и закрытия ее после завершения теста:

```
@pytest_asyncio.fixture(scope="function")
   async def test_session():
       session_maker.configure(bind=engine)
       async with session_maker() as session:
           try:
               yield session
           finally:
               await session.close()
```

Добавим фикстуру для создания клиента HTTPX:

```
@pytest_asyncio.fixture(scope="function")
   async def client(test_session):
       app.dependency_overrides[get_session] = lambda: test_session
       async with AsyncClient(
           transport=ASGITransport(app), base_url="http://localhost"
       ) as client:
           yield client
       app.dependency_overrides.clear()
```

Эта фикстура также переопределяет зависимость get_session для использования тестовой сессии базы данных.

Создание тестов для конечных точек REST API

Создадим файл test_login.py, в котором будем реализовывать тесты маршрутов аутентификации.

Начнем с импорта зависимостей:

```
import pytest
import httpx
```

Первой конечной точкой, которую мы будем тестировать, будет точка регистрации.

Добавим декоратор pytest.mark.asyncio, который сообщит pytest, что он должен рассматривать это как асинхронный тест.

Определим функцию и полезную нагрузку запроса:

```
async def test_sign_new_user(client: httpx.AsyncClient) -> None:

    payload = {
        "email": "testuser@server.com",
        "password": "testpassword",
    }
```

Определим заголовок запроса и ожидаемый ответ:

```
headers = {"Accept": "application/json", "Content-Type":
"application/json"}

    test_response = {"message": "User successfully registered!"}
```

Теперь инициируем сам запрос:

```
response = await client.post("/user/signup", json=payload,
headers=headers)
```

И, наконец, проверим, был ли запрос успешным, сравнив ответы:

```
assert response.status_code == 200
    assert response.json() == test_response
```

Запустим тест:

```
pytest tests/test_login.py
============================ test session starts
============================
...
collected 1 item

tests/test_login.py .
[100%]

============================  1 passed in 0.56s
============================
```

Определим тест для маршрута регистрации. Начнем с определения полезной нагрузки запроса и заголовков, прежде чем инициировать запрос, как в первом тесте:

```
@pytest.mark.asyncio
async def test_sign_user_in(client: httpx.AsyncClient) -> None:

    payload = {"username": "testuser@server.com", "password":
"testpassword"}

    headers = {
        "accept": "application/json",
        "Content-Type": "application/x-www-form-urlencoded",
    }
```

Далее мы инициируем запрос и проверим ответы:

```
response = await client.post("/user/signin", data=payload,
headers=headers)
    assert response.status_code == 200
    assert response.json()["token_type"] == "Bearer"
```

Повторим запуск теста:

```
pytest tests/test_login.py
============================= test session starts ==============================
...
collected 2 items

tests/test_login.py ..                                                   [100%]

============================== 2 passed in 1.36s ===============================
```

Создадим новый файл test_routes.py. Во вновь созданном файле добавим импорта:

```
import httpx
import pytest
from auth.jwt_handler import create_access_token
from sqlalchemy.ext.asyncio import AsyncSession
from models.events import Event
from typing import AsyncGenerator
from datetime import datetime
```

Мы импортировали функцию create_access_token и модель Event. Поскольку некоторые маршруты защищены, мы будем генерировать токен доступа.

Создадим новую фикстуру, которая при вызове возвращает токен доступа:

```
@pytest.fixture
async def access_token() -> str:
    return create_access_token("testuser@server.com")
```

Создадим новую фикстуру, которая добавляет событие в базу данных. Это действие выполняется для проведения предварительных тестов перед тестированием конечных точек:

```
@pytest.fixture
async def mock_event(test_session: AsyncSession) -> AsyncGenera-
tor[Event, None]:
    new_event = Event(
        creator="testuser@server.com",
        title="Event title",
        date=datetime(2024, 8, 28, 14, 38, 4),
        description="Event description",
        tags=["tag1", "tag2"],
        location="Event location",
```

```
    )
    test_session.add(new_event)
    await test_session.commit()
    await test_session.refresh(new_event)
    yield new_event
    await test_session.rollback()
```

Далее напишем тестовую функцию, которая проверяет метод GET в маршруте /event:

```
@pytest.mark.asyncio
async def test_get_events(
    client: httpx.AsyncClient, mock_event: Event, access_token: str
) -> None:
    headers = {"Authorization": f"Bearer {access_token}"}
    response = await client.get("/event/", headers=headers)
    assert response.status_code == 200
    assert response.json()[0]["title"] == mock_event.title
```

В предыдущем блоке кода мы проверяем присутствует ли событие, добавленное в базу данных в фикстуре mock_event.

Далее напишем тестовую функцию для конечной точки /event/{id}:

```
@pytest.mark.asyncio
async def test_get_event(
    client: httpx.AsyncClient, mock_event: Event, access_token: str
) -> None:
    headers = {"Authorization": f"Bearer {access_token}"}
    response = await client.get("/event/1", headers=headers)
    assert response.status_code == 200
    assert response.json()["title"] == mock_event.title
```

Далее напишем тестовую функцию, которая проверяет метод POST в маршруте /event/new. Мы создадим полезную нагрузку запроса, которая будет отправлена на сервер, заголовки запроса, которые будут содержать тип содержимого, а также значение заголовка авторизации:

```
@pytest.mark.asyncio
async def test_post_event(client: httpx.AsyncClient, access_token:
str) -> None:
    payload = {
        "title": "Event title",
        "date": "2024-08-28T14:38:04",
        "description": "Event description",
        "tags": ["tag1", "tag2"],
        "location": "Event location",
```

```
    }
    headers = {
        "Content-Type": "application/json",
        "Authorization": f"Bearer {access_token}",
    }
    test_response = {"message": "Event created successfully"}
    response = await client.post("/event/new", json=payload,
headers=headers)
    assert response.status_code == 200
    assert response.json() == test_response
```

Напишем тестовую функцию, которая проверяет метод PATCH в маршруте /event/edit:

```
@pytest.mark.asyncio
async def test_update_event(
    client: httpx.AsyncClient, mock_event: Event, access_token: str
) -> None:
    test_payload = {"title": "Updated event title"}
    headers = {
        "Content-Type": "application/json",
        "Authorization": f"Bearer {access_token}",
    }
    response = await client.patch("/event/edit/1", json=test_payload,
headers=headers)
    assert response.status_code == 200
    assert response.json()["title"] == test_payload["title"]
```

В предыдущем блоке кода мы изменяем событие, хранящееся в базе данных, затем мы определяем полезную нагрузку запроса и заголовки.

Напишем тестовую функцию, которая проверяет метод DELETE в маршруте /event/delete/{id}:

```
@pytest.mark.asyncio
async def test_delete_event(
    client: httpx.AsyncClient, mock_event: Event, access_token: str
) -> None:
    test_response = {"message": "Event deleted successfully"}
    headers = {
        "Content-Type": "application/json",
        "Authorization": f"Bearer {access_token}",
    }
    response = await client.delete("/event/delete/1",
headers=headers)
    assert response.status_code == 200
    assert response.json() == test_response
```

Чтобы убедиться, что событие действительно было удалено, добавим последний тест:

```
@pytest.mark.asyncio
async def test_get_event_again(
    client: httpx.AsyncClient, mock_event: Event, access_token: str
) -> None:
    headers = {"Authorization": f"Bearer {access_token}"}
    response = await client.get("/event/1", headers=headers)
    assert response.status_code == 404
```

Запустим все тесты, присутствующие в приложении:

pytest

```
============================= test session starts ==============================
...
collected 8 items

tests/test_login.py ..                                                   [ 25%]
tests/test_routes.py ......                                              [100%]

============================== 8 passed in 2.00s ===============================
```

ЗАКЛЮЧЕНИЕ

Фреймворк FastAPI предоставляет разработчикам мощный и гибкий инструмент для создания высокопроизводительных и масштабируемых API. Мы рассмотрели основные концепции и принципы работы с FastAPI, начиная от установки и настройки окружения до создания маршрутов и обработки запросов.

Одним из ключевых преимуществ FastAPI является его способность автоматически генерировать документацию для API, что значительно упрощает процесс разработки и тестирования. Благодаря использованию аннотаций типов и встроенной валидации данных, разработчики могут сосредоточиться на логике приложения, не беспокоясь о рутинных задачах. Кроме того, асинхронная природа FastAPI позволяет эффективно обрабатывать большое количество одновременных запросов, что особенно важно для современных веб-приложений с высокой нагрузкой.

FastAPI представляет собой современный и удобный инструмент для разработки программных интерфейсов веб-приложений, который сочетает в себе простоту использования и мощные возможности.

ОГЛАВЛЕНИЕ

Учебное электронное издание

ЕЛИСЕЕВ Алексей Игоревич
МИНИН Юрий Викторович

РАЗРАБОТКА ПРОГРАММНЫХ ИНТЕРФЕЙСОВ ВЕБ-ПРИЛОЖЕНИЙ С ИСПОЛЬЗОВАНИЕМ ФРЕЙМВОРКА FastAPI

Учебное пособие

Редактирование Е. С. Мордасовой
Графический и мультимедийный дизайнер Т. Ю. Зотова
Обложка, упаковка, тиражирование Т. Ю. Зотовой

ISBN 978-5-8265-2821-1

Подписано к использованию 16.10.2024.
Тираж 50 шт. Заказ № 111

Издательский центр ФГБОУ ВО «ТГТУ»
392000, г. Тамбов, ул. Советская, д. 106, к. 14
Тел./факс (4752) 63-81-08.
E-mail: izdatelstvo@tstu.ru